Towards Flexible Parsing of Structured Textual Model Representations

Dimitrios S. Kolovos[1], Nicholas Matragkas[2], and Antonio Garcia-Dominguez[3]

[1]Department of Computer Science, University of York, UK
dimitris.kolovos@york.ac.uk

[2]Department of Computer Science, University of Hull, UK
n.matragkas@hull.ac.uk

[3]Department of Computer Science, Aston University, UK
a.garcia-dominguez@aston.ac.uk

Abstract. Existing parsers for textual model representation formats such as XMI and HUTN are unforgiving and fail upon even the smallest inconsistency between the structure and naming of metamodel elements and the contents of serialised models. In this paper, we demonstrate how a flexible parsing approach can transparently and automatically resolve a number of these inconsistencies, and how it can eventually turn XML into a human-readable and editable textual model representation format for particular classes of models.

1 Introduction

Depending on the nature of a modelling language and the preferences of the engineers that work with it, models conforming to the language can be edited using graphical (e.g. diagram-based, tree-based, table-based), textual, or hybrid notations. Previous work has made the case for the value of text-based model representations as a means of leveraging existing powerful text editing tools and file-based version control systems [1]. In scenarios where text-based model editing is preferable for a new Domain-Specific Language (ie. metamodel), engineers are presented with two options.

- Define a bespoke textual syntax for the modelling language using frameworks such as Xtext, Spoofax, or EMFText;
- Use a generic syntax such as OMG's Human-Usable Textual Notation (HUTN) [2] or XML Metadata Interchange (XMI) [3].

The first approach can produce a tailored and concise syntax but requires substantial expertise and upfront development effort, and incurs a maintenance cost over time as the underlying framework/IDE evolves. The second approach on the other hand, does not require an investment in bespoke tool development, but comes at the cost of a more verbose and repetitive textual syntax. To a large extent, this is due to the rigidness of existing parsers for such generic textual

syntaxes. Existing parsers for XMI and HUTN require exact lexical equivalence between names defined in the metamodel (e.g. type and attribute names) and tokens found in serialised models and fail even upon the smallest inconsistency.

In this paper we propose a novel approach for parsing structured XML documents into in-memory models that conform to Ecore metamodels[1]. Instead of requiring exact lexical equivalence, the proposed approach uses approximate matching to map XML element and attribute names to metamodel types and their structural features. We use XML as an example structured format; the proposed approach is trivially portable to other textual formats such as HUTN, JSON and YAML.

The rest of the paper is organised as follows. Section 2 presents our flexible XML parsing algorithm with the help of a running example and discusses an open-source implementation of the algorithm (Flexmi) and supporting tooling, Section 3 presents the results of empirical evaluation experiments on the scalability and the practicality of the proposed algorithm, Section 4 discusses related work, and Section 5 concludes the paper and provides directions for further work on this topic.

2 Fuzzy Reflective Parsing

In this section, we present a novel flexible algorithm (parser) for parsing an XML document into an in-memory model that conforms to an Ecore metamodel. It is worth noting that the parser requires input XML documents to be well-formed[2].

The parser performs a depth-first traversal of the elements of the XML document, in which each element is visited exactly once. Before encountering the root element of the document, the parser expects to find an *nsuri* processing instruction (see line 2 in Listing 1.2) that declares the unique identifier (namespace URI) of the Ecore metamodel that the model is an instance of. In its current form, the parser supports only models that conform to a single Ecore metamodel (EPackage), but it is trivially extensible to support models conforming to multi-EPackage metamodels.

The parser uses a (initially empty) stack, to keep track of its position during the depth-first traversal of the XML document. When the root element of the XML document is encountered, the parser computes the string similarity[3] of the tag of the element with all instantiable (i.e. non-abstract) EClasses in the metamodel and selects the EClass with the maximum similarity. An instance of that EClass is created, is set as a top-level element in the in-memory representation of the model, and is pushed to the stack.

The parser then proceeds in a recursive depth-first manner to visit the descendants of the root element. When it encounters a new XML element, it follows the algorithm displayed in Listing 1.1. When it encounters the closing tag of an

[1] Ecore is the metamodelling language of the Eclipse Modelling Framework.

[2] `https://www.w3.org/TR/REC-xml/#sec-well-formed`

[3] In this work we use Levenshtein's string editing distance [4] to compute string similarities, however, other algorithms can also be used for this purpose.

XML element, it pops the top item of the stack (as discussed in Listing 1.1, this may be a model element, a containment slot, or *null*). A step-by-step discussion of the execution of the algorithm on a sample XML document is provided below.

The algorithm presented in Listing 1.1 explains how XML elements are mapped to model elements but does not discuss how the attributes of XML elements are mapped to attributes and non-containment references of the respective model elements; this is delegated to the *set_ attribute_ values* function which is invoked in lines 23 and 36, and is presented in Listing 1.3. A running example follows in Section 2.1.

Listing 1.1. Algorithm executed when the start tag of a new XML element is encountered by the parser

```
1  procedure fuzzy_parse(xml_element, stack)
2     let parent = stack.peek()
3     if parent is a model element then
4        let parent_model_element = parent
5
6        if xml_element has no attributes and only text then
7           /* xml_element is interpreted as an attribute value */
8           let attr = the attribute of the parent_model_element's EClass with the
                highest string similarity to the name of the xml_element
9           let parent_model_element.attr = textual content of the xml_element, cast
                appropriately
10          push null to the stack
11       end
12       else if xml_element has no attributes then
13          /* xml_element is interpreted as a containment slot */
14          let reference = the containment reference of the parent_model_element's
                EClass with the highest string similarity to the tag of the
                xml_element
15          push a containment slot that encapsulates the parent_model_element and
                the reference to the stack
16       end
17       else /*The element has attributes*/
18          /* xml_element is interpreted as a model element */
19          let candidate_types = all types of parent_model_element's containment
                references, and their sub-types
20          let new_model_element_type = the EClass from candidate_types with the
                highest string similarity to the name of the xml_element
21          let reference = the first reference of the EClass of parent_model_element
                that has a compatible type with new_model_element_type
22          let new_model_element = new instance of new_model_element_type
23          call set_attribute_values (new_model_element, xml_element)
24          if the reference is single-valued then
25             set the value of the reference to new_model_element
26          else
27             add new_model_element to the list of values of the reference
28          end if
29          push new_model_element to the stack
30       end
31    else if parent is a containment slot then
32       /* xml_element is interpreted as a model element */
33       let containment_slot = parent
34       let new_model_element_type = the EClass with the highest similarity to the
             tag of the xml_element, chosen from the EClasses that are subtypes of
             the type of the reference of containment_slot (including the reference
             type itself )
35       let new_model_element = new instance of new_model_element_type
36       call set_attribute_values (new_model_element, xml_element)
37       if the containment_slot's reference is single-valued then
38          set the value of the reference of containment_slot's model element to
                new_model_element
39       else
```

```
40            add new_model_element to the list of values of the reference
41        end
42    end
43  end procedure
```

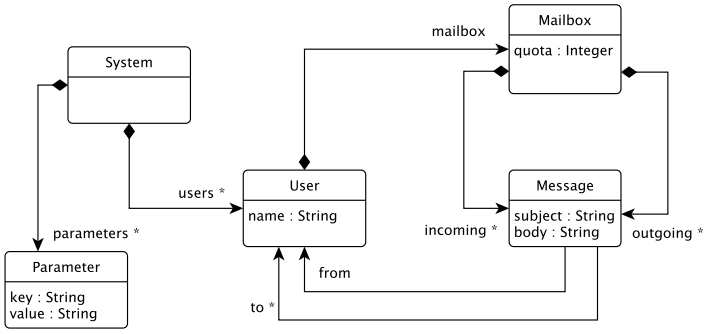

Fig. 1. Messaging System Metamodel

2.1 Running Example

This section presents the application of the algorithm of Listing 1.1 on the XML document illustrated in Listing 1.2. The Ecore metamodel that corresponds to the *http://messsaging* namespace URI in line 2 of the XML document is presented in Figure 1. During its application, the algorithm performs the following steps:

Listing 1.2. Example XML document

```
1  <?xml version="1.0"?>
2  <?nsuri http://messaging?>
3  <sys>
4    <u name="tom">
5      <box q="15">
6        <in>
7          <msg from="mary" subject="Hello␣Tom">
8            <body>
9              Fuzzy parsing is
10             so cool.
11           </body>
12         </msg>
13       </in>
14     </box>
15   </u>
16   <u name="mary">
17     <box q="20">
18       <out>
19         <msg to="tom,␣irene" t="Hello␣everyone"/>
20       </out>
21     </box>
22   </u>
23 </sys>
```

1. In line 2 of the XML document, the parser encounters the *nsuri* processing instruction, through which the XML document declares that it should be parsed as an instance of the *http://messsaging* Ecore metamodel (illustrated in Figure 1);
2. In line 3, the parser encounters the *sys* element. As discussed above, since the parser's stack is empty, it knows that it is at the root of the document and as such it calculates the similarity of the tag name (*sys*) against the names of all instantiable classes in the metamodel and decides that *System* is the closest match. As such, it creates an instance of *System*, places it as a top-level element of the model and pushes it to the stack.
3. In line 4, the parser encounters the *u* XML element. It peeks at the stack and sees that the top item is a model element (the instance of *System* that was created in the step above). Since the new XML element has attributes and complex children, as specified in line 39 of Listing 1.1, the parser attempts to convert it into a new model element. Since *u* is contained under a *System* element, according to the metamodel, its only two possible types are *Parameter* and *User*. Using string similarity, the parser decides that *u* is closer to *User*, creates an instance of the latter, adds it to the *users* reference of the *System* instance, and pushes it to the stack;
4. In line 5, the parser encounters the *box* XML element and behaves similarly to step 3: it creates an instance of *Mailbox*, adds it to the values of the *mailbox* reference of the *User* created in step 3, and pushes it to the stack. Here it's worth noting that since according to the metamodel only instances of *Mailbox* can be contained under instances of *User*, the actual tag of the element (*box*) is irrelevant. As such, if the *Mailbox* was to be renamed in an evolved version of the metamodel (e.g. to *Channel*), the parser would still construct a valid model;
5. In line 6, the parser encounters the *in* XML element. It peeks at the stack and sees that the top element is a model element (the instance of *Mailbox* created in the previous step). Since the *in* element has complex children but no attributes, according to line 12 of Listing 1.1, it treats it as a containment slot. As such it compares its tag name against the names of containment references in the *Mailbox* type (*incoming* and *outgoing*) and selects *incoming* as the closest match. It then encapsulates the *incoming* reference and the *Maibox* element into a *containment slot* object and pushes the latter to the stack;
6. In line 7, the parser encounters the *msg* element. It peeks at the stack and sees that the top element is the containment slot pushed in the previous step. As such, according to line 31 of Listing 1.1, it treats the *msg* element as a model element that needs to be added to the containment slot (i.e. to the *incoming* messages of the *Mailbox* created in step 4). Similarly, to step 4, the tag name of the XML element is irrelevant since there is only one type of model elements that can be contained under the *incoming* reference. As such, if the name of the *Message* class was to change in a revised version of the metamodel, the parser would still produce a valid model;

7. In line 8, the parser encounters the *body* XML element. It peeks at the stack and sees that the top element is the *Message* model element pushed in step 7. Since the *body* element does not have attributes and only has textual content, the parser treats its content as an attribute value. It identifies the actual attribute to be populated (*body*) by comparing the string similarity of the tag of the element against the names of all attributes in the *Message* class. Finally, it pushes *null* to the stack as no further elements exist under the *body* element;
8. In lines 11-15 the parser encounters the close tags of the elements started in lines 4-8 and for each close tags it pops the top item of the stack. After these lines are processed the top element of the stack is the instance of *System* pushed in step 2;
9. Lines 16-22 are processed in a very similar manner to lines 4-15, with the main difference being that the *out* element in line 18 is interpreted as a containment slot for the *outgoing* reference of class *Mailbox* – in contrast to step 5 where the *in* element was interpreted as a slot for the *incoming* reference.

2.2 Processing XML Attribute Values

Once the parser has determined that an XML element needs to be transformed into a model element, beyond instantiating the new model element, it also attempts to use the attributes of the XML element to populate the values of EAttributes and non-containment EReferences of the new model element. The first step of this process involves computing an optimal assignment of XML attributes to EAttributes and EReferences of the new model element (i.e. deciding which attribute will be mapped to which feature). This is an instance of the *assignment problem* – as every attribute in the XML element needs to be assigned to at most one EAttribute or EReference of the new model element – which the parser solves using the Hungarian method [5].

Once the optimal assignment between XML attributes and EAttributes/EReferences has been identified, the next step is to use the values of the prior to populate the values of the latter. This is achieved using the algorithm in Listing 1.3. Briefly, for multi-valued features the value of the XML attribute is first split using comma (,) as a separator and is then processed as follows. If the mapped feature is an EAttribute, then each fragment is cast to the datatype of the EAttribute and is set/added to the latter's value. For example when the algorithm processes line 17 of the Listing 1.2, it matches the *q* attribute to the *quota* feature of EClass *Mailbox*, it converts the value of *q* into an integer, and it then assign the *quota* attribute of the mailbox to it. Type casting errors are recorded so that they can be reported to clients of the algorithm.

If the mapped feature is an EReference, the algorithm temporarily records an unresolved reference. All unresolved references collected at this stage are attempted to be resolved after the entire XML document has been processed by the algorithm. References are resolved by matching against the IDs of model elements of compatible types. The ID of a model element is determined as follows:

– If the EClass of the model element has an EAttribute marked as *ID*[4], the ID of the model element is the value of that EAttribute;
– Else the ID of the model element is the name of its *ID* or *name* EAttribute (if one exists);
– In all other cases the model element is assigned an internal ID and cannot be referenced by other elements.

Listing 1.3. Algorithm that maps XML element attribute values to values of EAttributes and non-containment EReferences

```
1  procedure set_attribute_values(xml_element, model_element)
2     let assignment = the assignment with the maximum total similarity
3     foreach xml_attribute of the xml_element do
4        let feature = the corresponding feature of the xml_attribute in the
              assignment
5        if feature is not null then
6           let value = the value of the xml_attribute
7           if feature is an EAttribute then
8              if the feature is single-valued then
9                 set the value of the feature to value (typecast)
10             else
11                let values = split value by comma
12                set the value of the feature to values (typecast)
13             end
14          end
15          else /* the feature is an EReference */
16             if the feature is single-valued then
17                create an unresolved reference
18             else
19                let values = split value by comma
20                foreach value in values do
21                   create an unresolved reference
22                end
23             end
24          end
25       end
26    end
27 end procedure
```

2.3 Implementation

We have implemented the flexible reflective parsing algorithm discussed above on top of the Eclipse Modelling Framework, in the form of an implementation of EMF's *Resource* interface. The implementation (Flexmi) is available as open-source software under the permissive Eclipse Public License under `https://github.com/kolovos/flexmi`. Flexmi also comprises a dedicated editor which provides syntax highlighting and error reporting facilities.

3 Evaluation

Having presented the flexible reflective parsing algorithm in Section 2, in this section we report on the results of benchmarking the Flexmi prototype that implements it. Through this evaluation, we wish to assess 1) how the algorithm

[4] This is a modifier provided by Ecore, which is orthogonal to the name of an EAttribute

scales with increasing XML document sizes, and 2) whether the overhead imposed by the algorithm is prohibitive for practical use.

To conduct this evaluation, we implemented a model generator that produces different sizes of synthetic XML documents that are meant to be parsed as Ecore metamodels. We used the generator to produce models containing between 217 and 218,338 model elements (EPackages, EClasses, EAttributes, EReferences and EDataTypes), linked through both containment and non-containment references. We then transformed the generated models into an XMI representation so that we can compare the performance of the proposed flexible parsing algorithm with that of the EMF-based XMI parser in order to assess the magnitude of the overhead imposed by the nature of flexible parsing.

The results obtained through our benchmarks are displayed in the line chart of Figure 2. The x-axis of the chart represents the number of model elements, while the y-axis represents the time (in miliseconds) that the parsers required in order to parse models containing these elements. We can observe that Flexmi 1) scales linearly with the size of the model, and 2) is around 5.2 times slower than the XMI parser that ships with EMF.

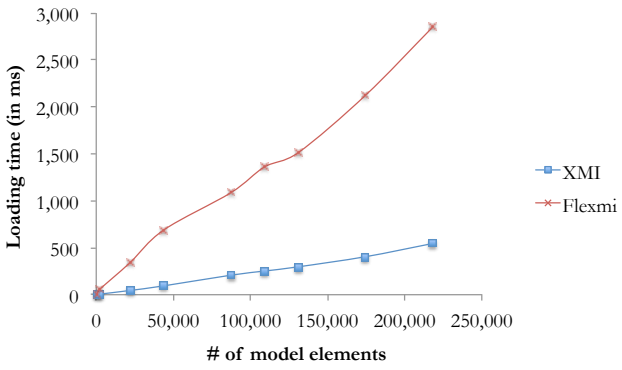

Fig. 2. Benchmark Results: Flexmi vs. XMI

3.1 Threats to Validity

For this empirical evaluation, we have used synthetic models that conform to an existing metamodel (Ecore). Conceivably, the performance of the algorithm on manually crafted models conforming to different metamodels could differ although, analytically, we see no reason for this. Evaluating the algorithm on manually crafted models is currently not feasible due to the unavailability of an existing data set, as this is a newly introduced approach to model parsing.

4 Related Work

While we are not aware of prior work that addresses the problem of loading models in a flexible manner in the context of MDE, repairing and recovering from

parsing errors is also studied in the XML community, where parsing is a core operation performed before an XML document can be navigated, queried, and manipulated. Generally speaking we can classify errors in XML documents in two main categories, namely well-formedness violations and structural invalidities. A well-formedness violation occurs when an XML document does not conform to the syntax production rules and the constraints defined in the XML specification [6], while structural invalidity occurs when the tree corresponding to an XML document is not accepted by the automaton associated with its schema.

Verifying well-formedness of XML is straightforward to do by using a stack in linear time. Automatically repairing a malformed XML document is more challenging though. In [7] the authors propose a dynamic programming algorithm which computes the edit distance between a malformed and its corresponding well-formed document. Such approaches can be complementary to our approach, since we assume that the input XML documents are well-formed.

On the other hand verifying and repairing structural invalidities is more demanding. First the similarity between the XML document and its corresponding schema must be calculated. Then the minimum number of tree edit operations, which make the document under consideration valid with respect to the schema, is calculated. Finally, generation of edit sequence operations is performed.

In [8, 9] an algorithm is presented for measuring the similarity between an XML document d and its corresponding schema D. This is done by measuring the common and the extra fragments between the two. In [10] the edit distance is defined as the minimum cost of all edit sequences transforming d into a valid document. A detailed survey of the different approaches for measuring similarity between an XML document and its schema is provided in [11].

The problem for measuring the minimum number of tree edits for making a tree valid with respect to a tree grammar is described for the first time in [12]. However, the algorithm proposed in this paper applies only to ranked trees. which is not the case of XML trees. In [13] the authors propose an algorithm for structural repairs of XML document with respect to single type tree grammars. Finally, [13, 14] propose an algorithm for automatically computing a set of edit operation sequences between an XML document and a tree grammar.

5 Conclusions

In this paper, we have presented a novel flexible reflective algorithm that can parse XML documents into in-memory models conforming to an Ecore metamodel. We have also discussed an open-source implementation of the algorithm and used it to evaluate its efficiency against the XMI parser provided by EMF. The results of our empirical evaluation show that the proposed algorithm scales linearly with the size of the XML document and that its overhead of flexible parsing is not prohibitive for practical use.

The main limitation of the proposed approach, in its current form, is that models loaded using Flexmi cannot be programmatically modified and persisted again as the parser does not memorise the mapping between XML element and

attribute names and the model elements/structural feature values it has produced from them. We plan to add such memorisation capabilities in future iterations of this work. We also plan to systematically investigate additional string similarity algorithms, beyond [4], and assess their suitability for flexible parsing.

Acknowledgements. This work was partially supported by the European Commission, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125).

References

1. Ole Lehrmann Madsen and Birger Møller-Pedersen. A Unified Approach to Modeling and Programming. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.
2. Object Management Group. Human-Usable Textual Notation Specification, 2004. `http://www.omg.org/spec/HUTN/1.0/`.
3. Object Management Group. XML Metadata Interchange (version 2.5.1), 2015. `http://www.omg.org/spec/XMI/2.5.1/`.
4. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
5. H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.
6. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical Report https://www.w3.org/TR/REC-xml/, W3C, February 2013.
7. Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. *Proceedings of the VLDB Endowment*, 6(9):601–612, 2013.
8. Elisa Bertino, Giovanna Guerrini, and Marco Mesiti. A matching algorithm for measuring the structural similarity between an xml document and a dtd and its applications. *Information Systems*, 29(1):23–46, 2004.
9. Elisa Bertino, Giovanna Guerrini, and Marco Mesiti. Measuring the structural similarity among XML documents and DTDs. *Journal of Intelligent Information Systems*, 30(1):55–92, 2008.
10. Slawomir Staworko and Jan Chomicki. Validity-sensitive querying of XML databases. In *Current Trends in Database Technology–EDBT 2006*, pages 164–177. Springer, 2006.
11. Joe Tekli, Richard Chbeir, Agma Traina, and Caetano Traina. XML document-grammar comparison: related problems and applications. *Open Computer Science*, 1(1):117–136, 2011.
12. Utsav Boobna and Michel de Rougemont. Correctors for XML data. In *Database and XML Technologies*, pages 97–111. Springer, 2004.
13. Martin Svoboda and Irena Mlỳnková. Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. In *Networked Digital Technologies*, pages 179–194. Springer, 2011.
14. Nobutaka Suzuki. Finding K Optimum Edit Scripts between an XML Document and a RegularTree Grammar. In *Proc. Emerging Research Opportunities for Web Data Management*, 2007. `http://ceur-ws.org/Vol-229/`.